AntiPatterns

Edward Jimenez

ekjimenez@gmail.com

Date: 04/24/2006

Abstract

As the concept of Design Patterns gains wider interest, it has become clear that patterns describing unsuccessful behavior can be equally enlightening. These “AntiPatterns” examine repeated failures looking to model what doesn’t work and then provide templates to refactor the problem into a more desirable solution. Because AntiPatterns start with a more “real-world” failed solution (rather than a Pattern, which starts with a problem and works towards and abstract goal solution), AntiPattern proponents find the study and implementation of AntiPatterns more applicable to many of today’s challenges.

In this paper, the author provides an introduction to AntiPatterns and describes the fundamentals of using them. Finally, an example is provided to show how AntiPatterns can be successfully applied in the real world.

TABLE OF CONTENTS

1. Introduction

The technique of searching for patterns within successful endeavors and then trying to emulate those patterns is not new. In 1977, architect Christopher Alexander and his colleagues introduced design patterns and pattern language in two books, A Pattern Language [1] and The Timeless Way of Building [2]. Alexander’s method introduced an innovative way to capture domain expertise for building architecture and town planning. He stressed that consistent solutions exist across variable problem sets.

> *“A pattern language is nothing more than a precise way of describing someone’s experience...”*
>
> -Christopher Alexander

Alexander referred to a group of patterns that share a particular context as a Pattern Definition Language. His concept was focused on building architecture and design with a stated purpose to create a non-technical vocabulary. However, several years later, the technology community would come to understand the implication of Design Patterns in a technological context.

In 1987 his work was rediscovered and began to be applied to the field of software development [3]. However, it was not until the mid 90’s that the use of patterns became

more prevalent and expanded to include areas such as software architecture and design, and organizational processes. In 1994 an industry conference on software design patterns, called Pattern Languages of Program Design (PLoP), featured several speakers, patterns, and pattern languages. Shortly after the conference, Erich Gamma et all (now known as the "Gang of Four") published the classic text, "Design Patterns: *Elements of Reusable Object-Oriented Software,"* which rallied object–oriented software architects around design patterns [1]. The Gang of Four's mantra was to "Reuse proven good design."

The concept and value of studying patterns of unsuccessful behavior, called “AntiPatterns,” while discussed in general early in the formation of Design Patterns, was not described formally until the mid 1990s. These Patterns examine repeated failures looking to model what doesn’t work and then provide templates to refactor the problem into a more desirable solution. Unlike Alexander with Design Patterns, there was no single seminal work attributed to the introduction of AntiPatterns.

Today, recommended solutions for organizations are often called “Best Practices.” In Project Management, the development of the Project Management Institute’s (PMI) Project Management Body of Knowledge (PMBOK) provides another example where the analysis of effective processes is used as a foundation for future success. While these examples resemble Design Patterns, there is an important distinction. That is, Design Patterns must use templates that should be consistent across the problem space.

AntiPatterns can currently be found across a range of disciplines including:

- Software Development
- Software Architecture
- Project Management
- Technology such as J2EE, Service Oriented Architecture, etc.

Just as Alexander's Design Patterns, AntiPatterns are maintained in catalogs called Pattern Definition Languages. An important principle of any pattern definition language regardless of its discipline is the consistent use of templates. As will be shown in Section 3, Elements of an AntiPattern, templates are of utmost importance to provide a consistent way to describe the Patterns.

2. Design Patterns versus AntiPatterns

Even though AntiPatterns were developed from the notion of Design Patterns, there is a significant difference between the two concepts. Those differences can be summarized into three fundamental categories:

- Area of focus
- Starting Point

- Output

Table 1 shows a comparison of the three aspects of Design Patterns and AntiPatterns. A description of those aspects follows.

	Design Patterns	AntiPatterns
Focuses on	Successes	Mistakes
Starting Point	Well-defined Question/Problem-based	Poorly Defined Solution-based
Solution Maps To	Unique Instance	Recommended Path

Table 1. Design Patterns vs. AntiPatterns

2.1 Area of Focus

The development of design patterns stems from the analysis of successful behavior. Design Pattern authors study successful solutions to articulate a repeatable path from the problem to the solution. AntiPattern authors study failed solutions, working backwards to determine common mistakes made.

To say that AntiPatterns focus on *mistakes* is not to say that the goal of AntiPatterns is to identify them. The expressed goal of AntiPatterns is to describe a way to refactor the failed solution into a successful one.

2.2 Starting Point

Figure 1, shows a diagram of the conceptual differences between Design Patterns and AntiPatterns. The starting point of the two illustrates an important distinction. Users of Design Patterns typically begin with a well-defined problem and in finding the relating pattern, identify the best path to take to the solution. AntiPatterns begin with a failed solution. Because the failed solution and the elements surrounding it are often ill-defined, the fundamental step of identifying the AntiPattern to use can be very challenging. Matters are further complicated by the fact that:

1. AntiPatterns are rarely isolated; and,
2. AntiPatterns with same causes but different symptoms can occur at the same time.

2.3 End Result

The goal of a Design Pattern is to map a recurring successful solution to a problem based on a certain context and outside forces. Design Patterns are generally designed to map to one unique solution. AntiPatterns on the contrary map to a refactored solution that is not

guaranteed to be unique [1].

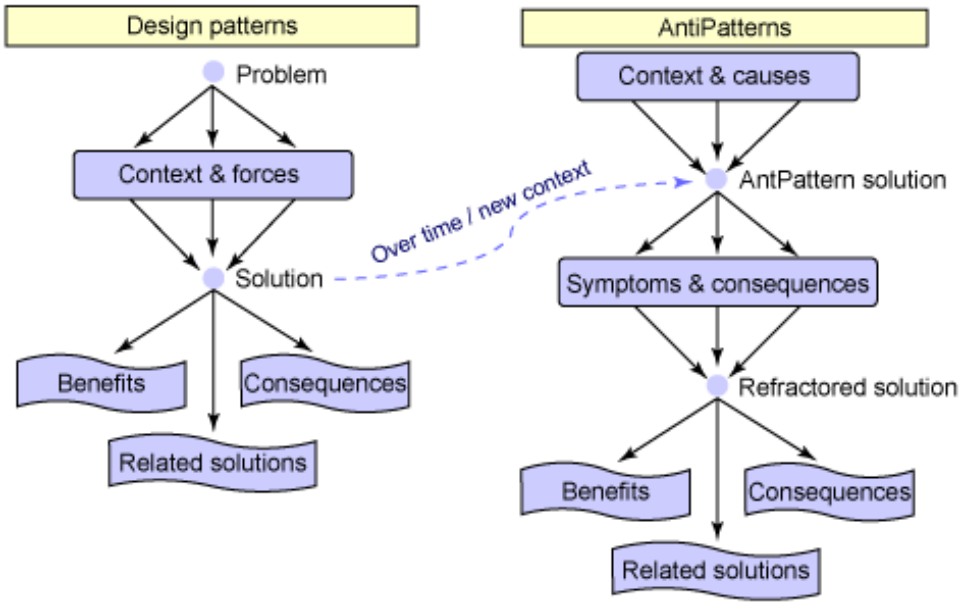

Figure 1. Design Pattern and AntiPattern Concept [1]

3. Elements of an AntiPattern

One of the most important aspects of using Design Patterns and AntiPatterns is the use of templates. They provide a consistent way to describe the Pattern, its causes, symptoms and resolutions. Templates also perform the important function of showing the likely consequences if the Pattern is not corrected. This can be a valuable way to justify the implementation (or lack of implementation) of the recommended refactoring approach.

There is not one single template that has been agreed upon by the AntiPattern community. However, most of the templates have several elements in common:

- **Pattern Name:** A good pattern name should possess two qualities:

    1. It should me memorable.
    2. It should be descriptive

  These qualities serve an important purpose. As discussed in the next Section, one of the first key steps to using patterns of any kind is the ability to find the right starting point. The slightly bottom-up approach often taken to match a failed solution to its appropriate AntiPattern can require the review of many templates. The ability to focus efforts on the most relevant Patterns makes the process much more efficient.

- **General Form:** The General Form section provides the general characteristics of the Pattern and an overall description of how it becomes apparent. The refactored

solution directly addresses this section.

- **Symptoms and Consequences:** This section provides a bulleted list of the Pattern's symptoms and consequences that indicate and/or result from the manifestation of the AntiPattern. Note, that if an AntiPattern is caught early enough, this section becomes predictive and can help the user identify whether or not the eventual outcome makes the Pattern worth addressing.

- **Typical Causes:** This section is a bulleted list that provides the user with a list of typical causes for the failed solution. The user will then compare this list to the real-world events that lead up to the failed solution.

- **Known Exceptions:** This section is important to differentiate when a diagnosed AntiPattern may be acceptable. For example, a stovepiped system, which may be undesirable in a production environment, may be totally acceptable in a prototype or research and development environment. This section will also be consulted to determine whether or not the AntiPattern should be addressed.

- **Refactored Solution:** The primary purpose of using AntiPatterns is to get to the point where the user can identify the recommended method for refactoring the current failing situation into a more desirable one. This section describes one or more methods to obtain the desired solution. As stated previously, by presenting a mapping to several possible refactoring methods rather than a single desired approach, this section provides an important distinction between AntiPatterns and Design Patterns.

This list of sections above is only a sample of some of the typical sections found in AntiPattern templates. These templates can include any number of sections. Additional sections often found include:

- Variations - Provides a listing of known variations of the AntiPattern. This section helps lead the user to the best-fitting Pattern when more than one applies.
- Related Solutions - Lists other related AntiPatterns.
- Examples – Provides one or more examples of the Pattern.
- Background – Provides additional information, historical context, etc.

It is important for readability that regardless of the sections that are used, they should remain consistent throughout the pattern definition language.

4. Using AntiPatterns

AntiPatterns provide a succinct, easy-to-read way to describe and discuss a problem. Developing AntiPatterns can be an effective way to analyze a problem space and to capture expert knowledge.

The more utilitarian use of AntiPatterns as a tool for refactoring a failed solution can be broken into three fundamental steps.

Step 1: Understand the AntiPattern domain

Understanding the AntiPatterns that exist in a particular domain space is important for several reasons. First, a user may be deeply entrenched in an AntiPattern before perceiving a side effect. Knowing the symptoms of relevant AntiPatterns may make it possible to identify and address the causes before serious consequences arise.

Understanding the AntiPattern domain also helps the user find the most relevant AntiPattern (Step 2) as quickly as possible.

Step 2: Identify which AntiPattern is appropriate

Beginning with an overall understanding of the AntiPatterns and understanding the General Form of those AntiPatterns, the user begins to search for the most relevant AntiPattern. This is accomplished by comparing what is known about the existing failing solution to the General Form, Symptoms, Consequences and Typical Causes.

Step 3. Implement the most appropriate solution

The most important element of an AntiPattern is the existence of a refactored solution [1]. The Refactored Solution section of the Pattern describes one or more possible ways to refactor the current failed solution.

Note that every AntiPattern does not need to be addressed for the organization or project to be successful. To help evaluate the most appropriate course of action, the user should consider the consequences of the AntiPattern. Users should also review the Known Exceptions section to verify that the Pattern is still relevant in the given context.

In some cases, based on the factors above, related cost, risk, etc., the most appropriate course of action may be to do nothing.

5. Example Application

> *“The AntiPattern amplifies the problem in a way that helps organizations to recognize the problematic structure, symptoms, and consequences”.* [1]

An interesting challenge that arises when using AntiPatterns is to understand the potential cascading that can occur from one AntiPattern to another. Users of a Pattern can follow back through the symptoms to understand the steps that brought them to that point. As the users work backward, they may get to a point when the earlier symptoms and causes of the AntiPattern do not mesh with their experience. AntiPattern users must be able to

trace from that point to another Pattern.

The following example, diagramed in Figure 2, describes how an AntiPattern user can follow the causal flow back through the Patterns to arrive at the most relevant starting solution. The example then proceeds through the relevant tasks to illustrate the typical process used.

As described in Section 4, the first step in the process is to identify the appropriate Pattern. In this example, the user describes a developed application and initially identifies two characteristics of the existing failed solution: the application is costing more money than anticipated; and, the product isn’t properly interfacing with the existing software baseline.

The user begins by identifying a symptom, cost overruns. This initially leads the user to the Project Mismanagement AntiPattern. Further investigation shows another symptom, the application seems to be working incorrectly; it appears that the wrong product was developed. Note that focus is directed toward the project manager. However, the Pattern continues to describe other symptoms that do not apply. In this case, sufficient testing was performed and the testers documented their evaluation of design defects. The AntiPattern user decides that Project Mismanagement is not the correct Pattern.

Continuing, the user identifies the Stovepipe System AntiPattern. The symptoms evaluated thus far are the same, and the AntiPattern seems like a good fit. The development of the wrong product is explained by the gap between the developed software and the system architecture. Note the focus is now turned toward the System Architect. However, further investigation, shows that not only does the software meet the customer's expectations, but the developers and the architects agree that they have interpreted the design correctly and with the same vision. The user decides to pass over the Stovepipe System Pattern and continue looking.

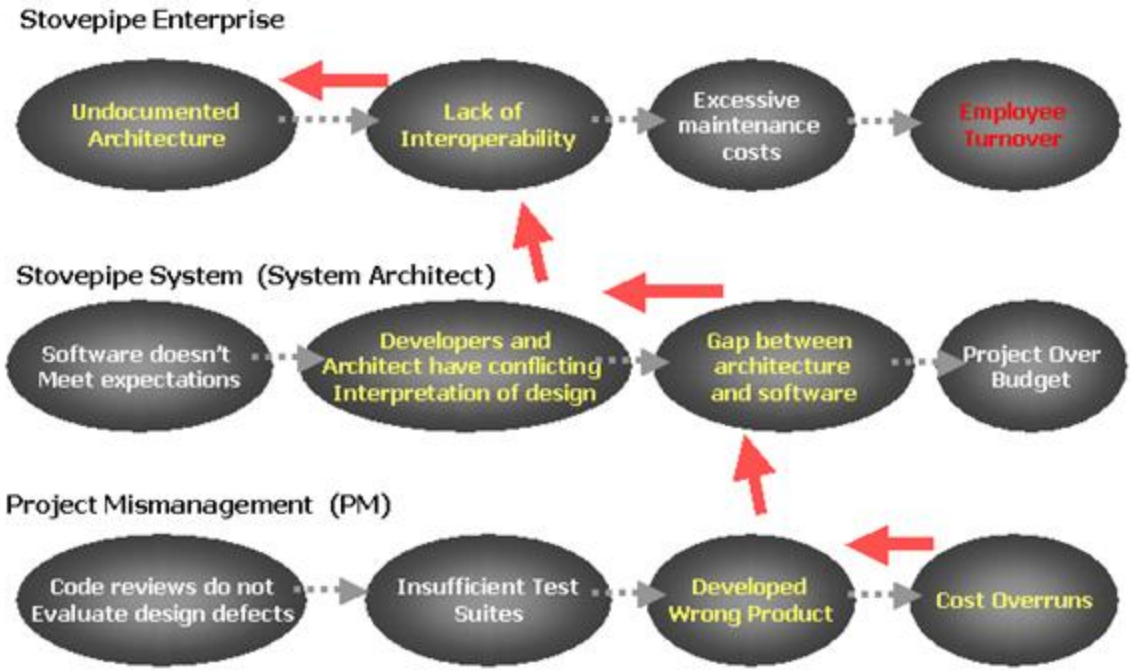

Figure 2. Example of the Cascading Nature of AntiPatterns

The user discovers that the enterprise does not have a documented architecture. Reviewing the Symptoms and Causes of the Stovepipe Enterprise Pattern, the user realizes that the apparent poor functionality of the application as well as the excessive costs are due to a lack of interoperability between systems. With this realization, the true AntiPattern origin is discovered: Stovepipe Enterprise.

To verify the general form, the user reviews the General Form and Symptoms. In this example, the relevant symptoms include:

- Incompatible terminology
- Incorrect usage of technology standard
- Excessive maintenance costs

The user now studies the AntiPattern Causes. This allows further verification and helps the user begin to expose possible problem areas. For this AntiPattern, possible causes include:

- Lack of standard reference model
- Lack of knowledge of technology standard used
- Absence of horizontal interfaces in systems integration solution

The user then turns to the Known Exceptions. While there are identified instances where this AntiPattern is acceptable, for this example, the user concludes that this case does not fall into one of those categories.

In order to decide whether or not to implement a refactored solution, the user turns to the AntiPattern Consequences and weighs the cost of inaction. In this example, consequences include the lack of software reuse. However, it was not until the second relevant consequence was discovered, high maintenance costs, that the decision was made to implement the refactored solution.

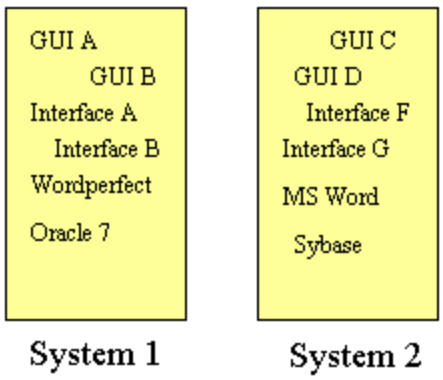

Figure 3. Initial Architecture (Starting Failed Solution)

Upon taking the decision to refactor, the user must now review the list of recommended refactor solutions and choose the one that is most appropriate. In this case, the solution selected is to refactor the stovepiped enterprise shown in Figure 3.

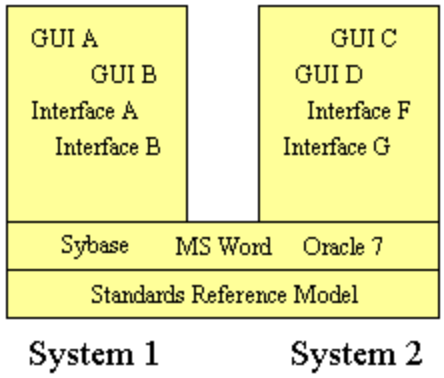

Figure 4. Refactored Architecture

The definition of the steps required to achieve the refactored solution is specifically described in the context of getting from the current condition (Figure 3) to the new desired solution (Figure 4). This refactored solution is achieved by following the well-documented tasks contained in that Pattern template. In this example, those tasks include defining a standard reference model, and identifying some baseline standards.

This example illustrates a more complicated use and nature of AntiPatterns. It illustrates how a user may need to trace backwards through a number of AntiPatterns (and across several different contexts) to find the most appropriate pattern.

6. Writing AntiPatterns

The authors of *AntiPatterns* stress that writing AntiPatterns should be fun. They should draw upon real-world experience and should follow a few fundamental guidelines. The authors suggest that AntiPatterns should:

1. Adequately and appropriately fit into the defined pattern language template,
2. Describe an observed repeating pattern; and,
3. Have a proven refactored solution.

> *"Without a solution, it's not an AntiPattern. It's just a rant ".* [1]

AntiPatterns can also be (but are not necessarily) defined across different perspectives. For example, one author defines the Patterns from three perspectives: 1) Manager; 2) Architect;

and 3) Developer.

AntiPatterns are not:

- Restricted to a particular context – AntiPatterns exist for project management, organizations, software development, etc.
- Untested ideas, theories or new inventions – AntiPatterns should be observed, repeating patterns with a proven solution for refactoring.
- A way to single out "the bad guy" (see Section 7.4).
- A "silver bullet" or panacea – AntiPatterns articulate ways to refactor bad solutions into good ones based upon previously captured expertise. While suggested methods of refactoring may exist, the recommended solution may be very difficult to implement.

7. Challenges for AntiPatterns

AntiPatterns continue to be a relatively immature technology. In the investigation for this paper, several challenges were identified. The author believes that addressing some or all of these challenges will help AntiPatterns gain acceptance.

7.1 Knowledge Base

As with Design Patterns, the strength of AntiPatterns comes from an expanded knowledge base. This typically relies on the communal nature of writing patterns and is facilitated by online resources such as, Wikis, blogs, etc. For AntiPatterns to become increasingly useful, it will be important to enhance the presence of this online collaboration.

While Design Patterns have proponent groups such as hillside.net, AntiPatterns have yet to receive that level of sponsorship.

7.2 Lexicon

It is accepted both with Design Patterns and AntiPatterns that there are many different pattern definition languages and corresponding templates. However, the overall lexicon of Design Patterns is generally consistent. This is not the case with AntiPatterns; possibly due to the lack of a single seminal work. Even the definition of AntiPatterns is inconsistent. For example, while one author includes refactoring as a most important part of AntiPatterns, another author describes AntiPatterns as the pattern that defines the problem. He uses the term Amelioration Pattern to describe the pattern required for refactoring. As AntiPatterns continue to develop, it would appear that their lexicon must standardize.

7.3 Organization

Organization of the AntiPattern repository is important. The more rich the set of AntiPatterns, the more difficult it can be to find the one that is most relevant to the issue at hand. Paradoxically, AntiPatterns must avoid an "Information Overload" Pattern.

In addition, AntiPatterns must be compartmentalized to facilitate focusing on the relevant AntiPattern. However, they must also be transparent enough so that a user is able to follow a cascading AntiPattern. In the example from Section 5, the user follows the AntiPattern cascade from a Project Mismanagement AntiPattern back through a Stovepipe Enterprise AntiPattern.

7.4 The Fall Guy

Because AntiPatterns focus on mistakes, a logical extension of that investigation is to search for the person who is responsible. Using AntiPatterns in this way creates unwilling participants and threatens effective use of the Patterns. In *AntiPatterns*, the authors stress this point: "While assigning blame and pointing fingers may provide a temporary rush of satisfaction, this is not the intended use of software AntiPatterns." [1]

It is also important to remember that AntiPatterns can arise through no fault of an individual. For example, obsolescence of technology can cause a successful pattern to become an AntiPattern (see Figure 1).

8. Related Word

There is no single group, book, or web site that is the definitive resource for AntiPatterns. However, the book referenced most often is "AntiPatterns, Refactoring Software, Architectures, and Projects in Crisis," by W. Brown et al. These authors have also written a book for AntiPatterns in project management. They are working to create a website to facilitate Pattern collaboration.

In January 2006, a new book on AntiPatterns was published by Phillip A. Laplante called "AntiPatterns: Identification, Refactoring, and Management."

Some authors of individual AntiPatterns can be found on the Web. Online, in the IBM Webservices library, an article by Jenny Ang describes an AntiPattern for Service Oriented Architecture:

http://www-128.ibm.com/developerworks/webservices/library/ws-AntiPatterns/

A new book by Martin L. Shoemaker that is yet to be released is entitled, "Requirements Patterns and AntiPatterns : Best (and Worst) Practices for Defining Your Requirements."

In addition, the Hillside Group is a non-profit organization that in addition to maintaining a large online pattern library (at http://hillside.net/patterns/) sponsors many conferences on design patterns. For several years, one of their annual conferences, the Pattern Language o